INTERNATIONAL JOURNAL OF SYSTEMS APPLICATIONS, ENGINEERING & DEVELOPMENT
DOI: 10.46300/91015.2021.15.7 Volume 15, 2021

Database Access Layer Code Generation Directly from Use Case Scenarios

[1]Nassima Yamouni-Khelifi, [2]Kaddour Sadouni, [3]Michał Śmiałek, [4]Mahmoud Zennaki
[1,2,4] University of Science and Technology-Mohamed Boudiaf-
El Mnaouar, BP 1505, Bir El Djir 31000, Oran, Algeria
[3] Warsaw University of Technology,
Pl. Politechniki 1, 00-661, Warsaw
Poland

Received: February 20, 2021. Revised: March 27, 2021. Accepted: March 30, 2021. Published: April 1, 2021.

Abstract— Requirements definition is the first step in the life cycle of a software system. Requirements are formulated as paragraphs of text and appear ambiguous, so they cannot be translated directly into code. For this reason, they are treated as secondary artifacts for software developers. This paper presents a model-driven based approach where requirements are treated as first-class citizens, and can contribute to the final code. In this approach, requirements are formulated as use case models with their textual scenarios, using a precise requirements language called RSL, allowing an automatic transition to executable Java code. The structure of the generated code follows the Model-View-Presenter (MVP) architectural pattern. The work focuses on the Model layer code, which is responsible for the persistence and storage of data in a database system.

Keywords— model-driven requirements engineering, use cases, scenarios, model transformation, metamodels.

I. Introduction

Software development starts with requirements definition, conceptual and design models, and ends with source code generation and maintenance. Requirements engineering defines the problem domain of a software system and determines the needs of users and the environment [1]. Errors at this stage can affect the other stages of the life cycle and the quality of the software [2]. The use cases invented by Ivar Jacobson [3] and their descriptions are widely used to specify functional requirements, they are written in natural languages (e.g. NLP techniques [4]), and using semi-formal diagrams such as the Object Management Group (OMG) Unified Modeling Language (UML) [5]. The use cases are therefore ambiguous, and their transition to design models and code is done manually [6].

Many researchers in this field have proposed alternative visual notations or extensions to other existing approaches (like UML language…), to treat use cases as first-class citizens in software development, and to automate the translation of these models and their descriptions into other models or code. Some relevant studies can be found in [7], [8], [9], and [10], …etc. Model-Driven Requirements Engineering (MDRE), officially introduced in 2001 at the first International Workshop on Model-Driven Requirements Engineering in San Diago [11], specifies requirements by developing requirement models. In this paper, another term is used, namely Requirements-Oriented Programming, which focuses on the direct contribution of the requirements models to the final code, and brings the programming activities closer to the precisely specified requirements [12].

Hermann Kaindl, Michał Śmiałek et al. in [13] introduced a new language for specifying requirements, called the Requirements Specification Language (RSL), implemented in the framework of ReDSeeDS [14]. This language provides more precision for use case notation and their representations (scenarios), and allows the automatic translation of use case models into more detailed models [15]. To our knowledge, no work has been introduced for the automatic translation of functional requirements (e.g., use case models) into database access code.

In this work, a set of translational semantics was developed for the RSL in terms of UML and Java code to manage database access: persisting and storing data in a database. The algorithms are expressed in a graphical transformation language which is MoLA to implement these semantics [16].

The effectiveness of the proposed method was validated with a case study for the "Tribunal E-Services" system.

Fig. 1 presents the proposal of a transformation process that allows to generate a database access code from high-level requirement models. The source model includes the use case diagrams and their textual scenarios written in RSL language. The target models are: model classes, and DTO classes written in Java. MoLA was used to perform the transformation. The mapping of the Object code to the Relational code was performed using the Hibernate Object Relational Mapping tool [17]. Hibernate allows to map persistent classes of Data Transfer Objects (DTOs) to database tables using XML files, and to manage the data using CRUD (Create/Read/Update/Delete) operations.

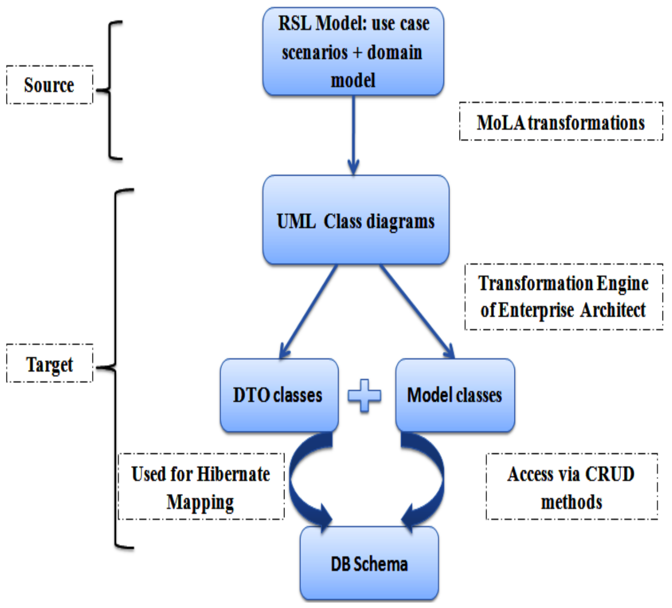

Source
RSL Model: use case scenarios + domain model
MoLA transformations
UML Class diagrams
Transformation Engine of Enterprise Architect
Target
DTO classes
Model classes
Used for Hibernate Mapping
Access via CRUD methods
DB Schema

Fig. 1 Transformation process

II. Definition of Use Cases

This section describes the RSL language used to define precise use cases and their descriptions: scenarios that are well linked to domain model elements. RSL introduces a comprehensive language that is based on natural language constrained sentences described in Subsection **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.**. The RSL use case constructs allow defining the entire application logic of a software system. Application logic refers to the observable behavior of the application as perceived by users, and covers the interactions between the user and the system. The following subsections provide details of RSL constructs that are related to the generation of source code from use case scenarios related to domain elements (notions). Details of other RSL constructs are beyond the scope of this study are discussed in [12].

A. Overview of Requirements Specification Language

The Requirements Specification Language (RSL) is a semi-formal language for specifying the requirements of a software system [7], [15], and [18]. RSL is different from other requirement languages because it allows to separate the description of the system behavior from the description of the problem domain, and to keep the relationship between them via "hyperlinks". System behavior can be described by thought use cases and their textual scenarios. The domain definition in RSL includes "actors", "system elements" and "notions" (words: "nouns" or "verbs"). These notions are used in scenario sentences that describe use cases. Thus, the basic constructions of the RSL allow the specification of typical text-based requirements specifications with some graphical elements. The RSL grammar is defined by the metamodel written in a meta language called Essential Meta Object Facility (EMOF) standardized by the Object Management Group (OMG) consortium [19], which is a subset of MOF^{TM}. The main objective of MOF is to allow the definition of metamodels using the basic syntax of class models (classes with attributes and relationships). The full description of the RSL language (abstract syntax, concrete syntax and semantics) can be found in [12], and [13]. Functional requirements in RSL are defined by use case models. Use cases are observable elements of functionality that lead to goals that may succeed or fail. Use cases are derived from the UML, but here they define new and changed features.

B. Domain elements

In RSL, all kinds of problem domains can be defined (physics, biology, aeronautics, etc.). Regardless of the domain, a set of related concepts and possible ways to visualize and process the data related to these concepts must be defined, which constitutes the domain (business) logic of the system. RSL offers three types for domain elements: *"Actors"*, *"SystemElements"*, and "*Notions*". Notions have names and may contain *"DomainStatements"*. A domain statement consists of a single *"Phrase"*. The concrete syntax of the domain statement is explained in Fig. 5.

Notions have several types; the basic element is the *"Concept"*. Its notation is equivalent to the UML class. The other types of elements are the *"Attributes"*. These elements are not contained graphically in concepts as in UML classes; they are separated from notions and include information about the type of data (text, real number, date, true/false, ...). Data types are not limited and can be extended according to the problem domain, but must be defined in advance during transformation. The other types of RSL are: *"Data Views"*. They are divided into two types: *"Simple Data Views"*, and *"List Data Views"*. Data views point to a set of attributes. "Simple Data Views" are used to present single instances of combined attributes. "List Data Views" are used to present lists containing many instances. These types of domain elements (notions) are linked by appropriate relationships (e.g., association, containment).

In this work, these data views are used in the transformation, and are mapped to views in the database system. Section III explains this process.

In addition to the business logic and its elements, the application logic in the RSL is defined by UI elements, which must be linked to the domain elements by relationships. RSL offers four types of UI elements: *"Screens"*, *"Triggers"*, *"Messages"* and *"Confirmations*". The most important elements are the "Screens" as they contain the other elements. The "Triggers" are associated with the user's interactions with the system. "Messages" and "Confirmations" are used to present information and accept user decisions. "Screens" can associate "Data Views" to present and update data, through *"present"* and *"update"* relationships. Triggers may point to a data view or an attribute to indicate the data that needs to be updated when they are invoked. "Screens" and "Triggers" are linked by the *"action param"* relationship, and are linked to the domain elements by arrows. These UI notions also contribute to the generation of code for graphical elements (see Section V). More details on the metamodel of UI elements and their relationships can be found in [12].

Fig. 2 illustrates part of the RSL metamodel for the notions and the relationships between them. The "Domain elements" can be connected by *"DomainElementRelationships*. One of the domain elements is treated as the *"source"* of the relationship, and the other as the *"target"*. The directed attribute indicates whether the domain element is a source element or a target element. Relationships have multiplicities: *"sourceMultiplicity"* and *"targetMultiplicity"*.

Attributes in RSL, are normal notions, and have "Data Type" defined via the *"primitiveDataType"* metaclass. The possible values of the data types are defined by an enumeration, as shown on the left side of Fig. 2. The concrete notation in Fig. 3 illustrates an example of an RSL domain model, which includes two concepts within their attributes. These concepts and attributes are presented by rectangles with appropriate tags, and are linked by a *"containment"* relationship, which resembles an *"aggregation"* relationship in UML. Concepts are interconnected by relationships of "association" of multiplicities type, from *"1"* to *"*" (many)*.

C. Constrained Language Sentences and Scenarios

Some domain rules need to be defined to complete the description of the RSL. This implies how to process the data?.

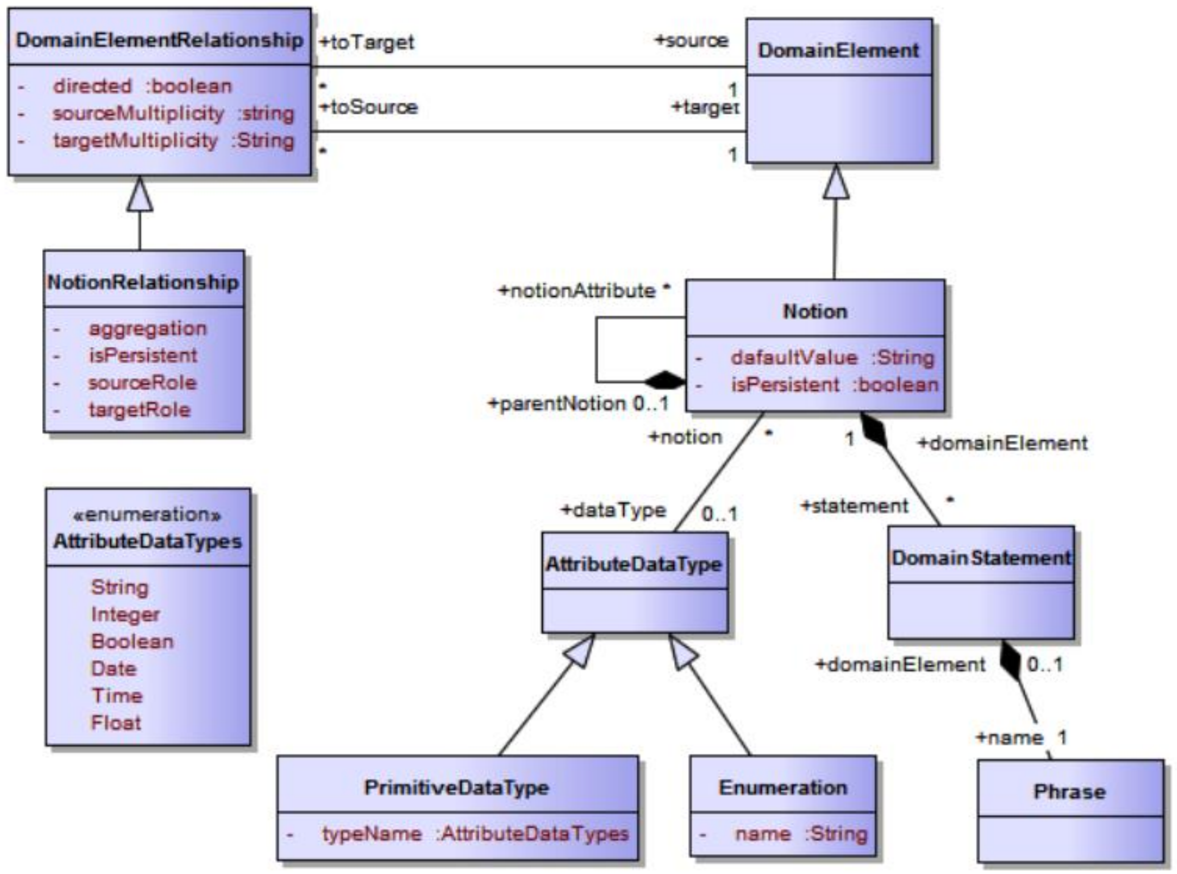


Fig. 2 Domain Model metamodel for Notions and their relationships

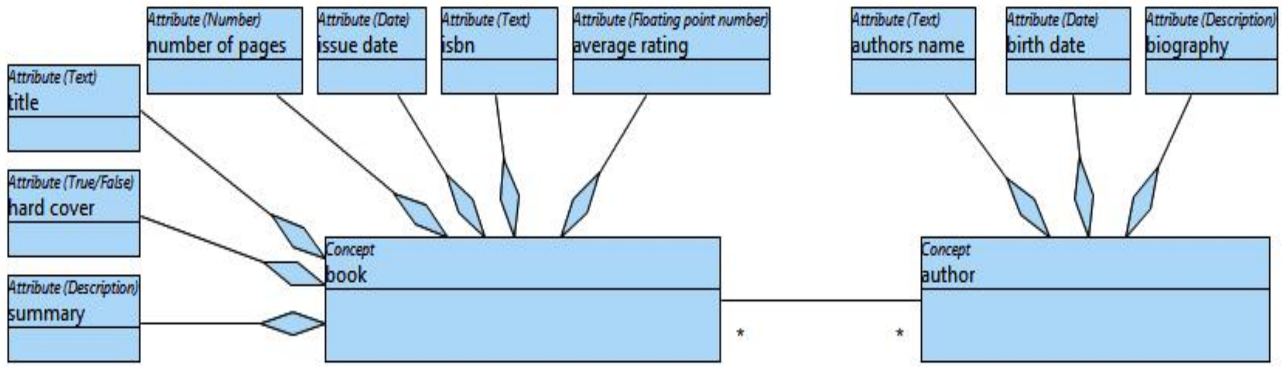


Fig. 3 Example of a concrete syntax of a domain model

In RSL, data is processed by means of *"verb phrases"*. Verb phrases are similar to operations in UML. The difference is that they have no parameters, and they have a defined verb-noun grammar. Verb phrases can be contained in most types of domain elements (concepts, screens, data views, etc.). There are various predefined types depending on the types of the domain elements. For problem domain elements (Concepts, and Data Views), the predefined actions available are the common CRUD operations (CREATE, READ, UPDATE, DELETE) and VALIDATE. The other predefined actions for the rest of the domain elements are not described here. For more information, the reader can refer to [12].

- **CREATE:** adds new items to the system. A set of verbs can be used as keywords for the "CREATE" action: "create", "save", "add" and "write".
- **READ:** obtains values from data items in the system. The following verbs can be used for "READ" action: "read", "get", "fetch", "retrieve", "search" and "build".
- **UPDATE:** substitutes data in the system with other new values. The following list of verbs can be used for "UPDATE" action: "update", "modify", "edit" and "override".
- **DELETE:** deletes data from the system. The following verbs can be used for "DELETE" action: "delete", "remove", "erase" and "destroy".
- **VALIDATE:** verifies the values of a domain element. The following verbs can be used for the "validate" action: "validate", "verify", "check", "inspect" and "examine".

Each RSL use case must have a main scenario and alternative scenarios that lead to the achievement or failure of the objective. The scenario (story) consists of a sequence of actions performed either by the user or by the system. The actions are expressed in simple sentences in the form of a Subject-Verb-Object (Indirect Object) grammar. The subject indicates who performs the action (user or system), the verb describes the operation that can be performed (e.g., build, show, search, etc.) and the objects represent notions (e.g., course list). The indirect object represents the detailed data transmitted when executing actions (e.g. with book list) [15], [18], and [20].

Fig. 4 illustrates the abstract syntax of SVO sentences in use case scenarios, which is composed of two metaclasses: "Subject" and "Predicate". These two metaclasses represent phrase hyperlinks, one pointing to: "NounPhrase" and the other to "VerbPhrase". These two hyperlinks point to phrases that contain the appropriate text. Each SVO sentence predicate is a hyperlink to a verb phrase. The object indicates the actual domain element, and the verb selects the appropriate verb phrase. In the RSL editor, these links must be maintained automatically. Each time an SVO sentence is created, its parts should be hyperlinked to the appropriate phrases in the domain model.

In RSL, there are certain types of constrained language sentences. This study only focuses on one type of sentences, namely SVO sentences. SVO (-O) sentences can also be divided into three main categories, depending on whether the subject refers to "Actor" or "System". Furthermore, six sub-categories can be distinguished. Two are "Actor-to" sentences and four are "System-to" sentences.

- **Actor-to-System:** the subject is an actor and the direct-object is an element of the user interface (button, option, etc.). Two subtypes can be distinguished: "Actor-to-Trigger" and "Actor-to-Data Views".
- **System-to-Actor:** the subject points to a system and the direct-object is a user interface element (window, form, etc.). There are also two subtypes: "System-to-Dialogue" and "System-to-Screen".
- **System-to-System:** the subject points to a system and the direct-object points to a domain element. This category has two types: "System-to-Concept" and "System-to-Data".

This study uses sentences from the sub-category "System-to-Data Views", which facilitate the management of data in a database system. Fig. 5 shows an example scenario for "E-Correction of Civil Documents" in the "Tribunal E-Services" use case model. In this concrete notation, as can be seen, the scenario is a sequence of textual SVO sentences, which are numbered. The number of sentences allows for better readability and referencing. The scenario contains ten SVO (-O) sentences, with four types of actions, one of which is a CRUD action("Update"), and the other is a "Validate" action.

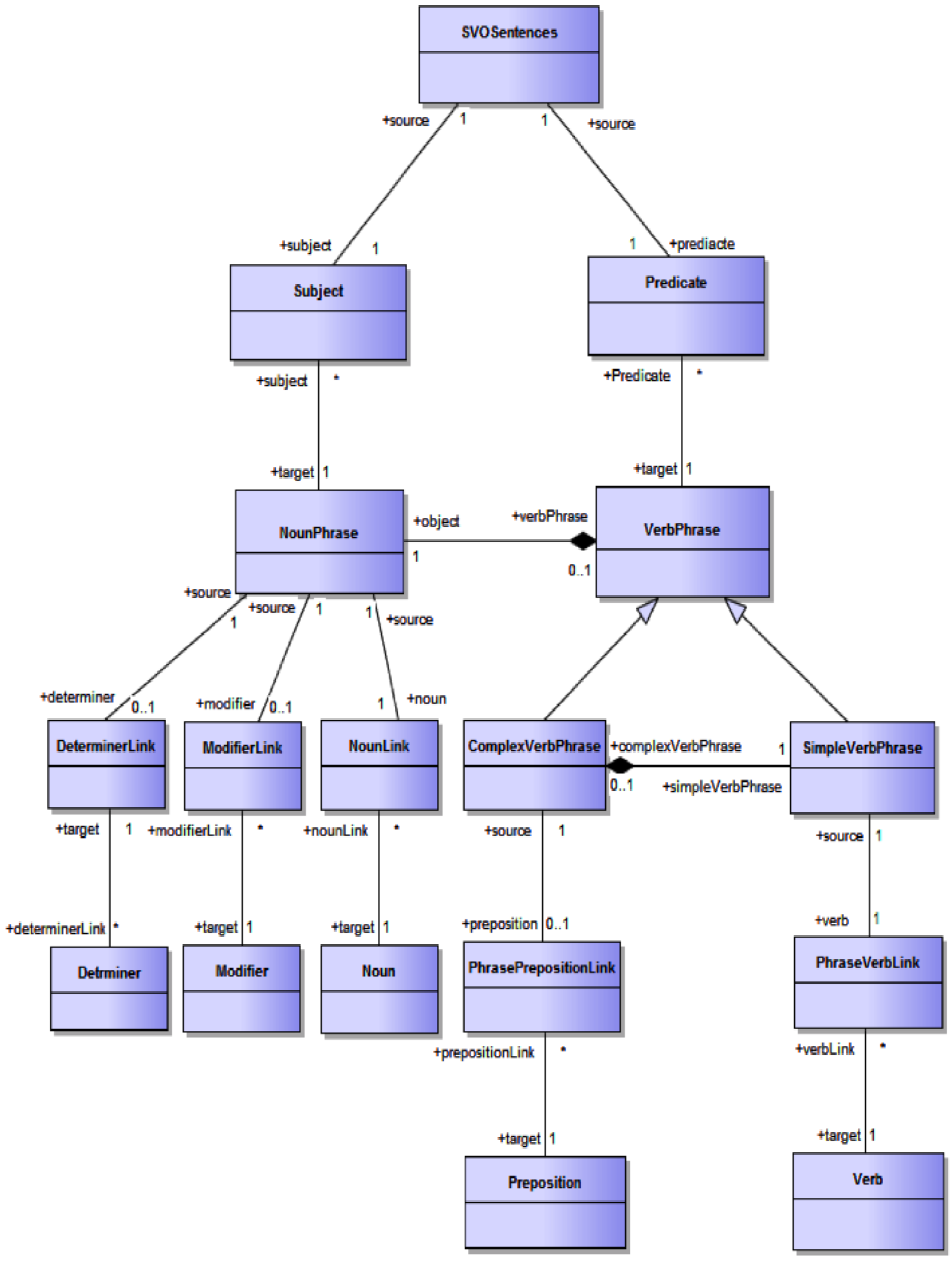
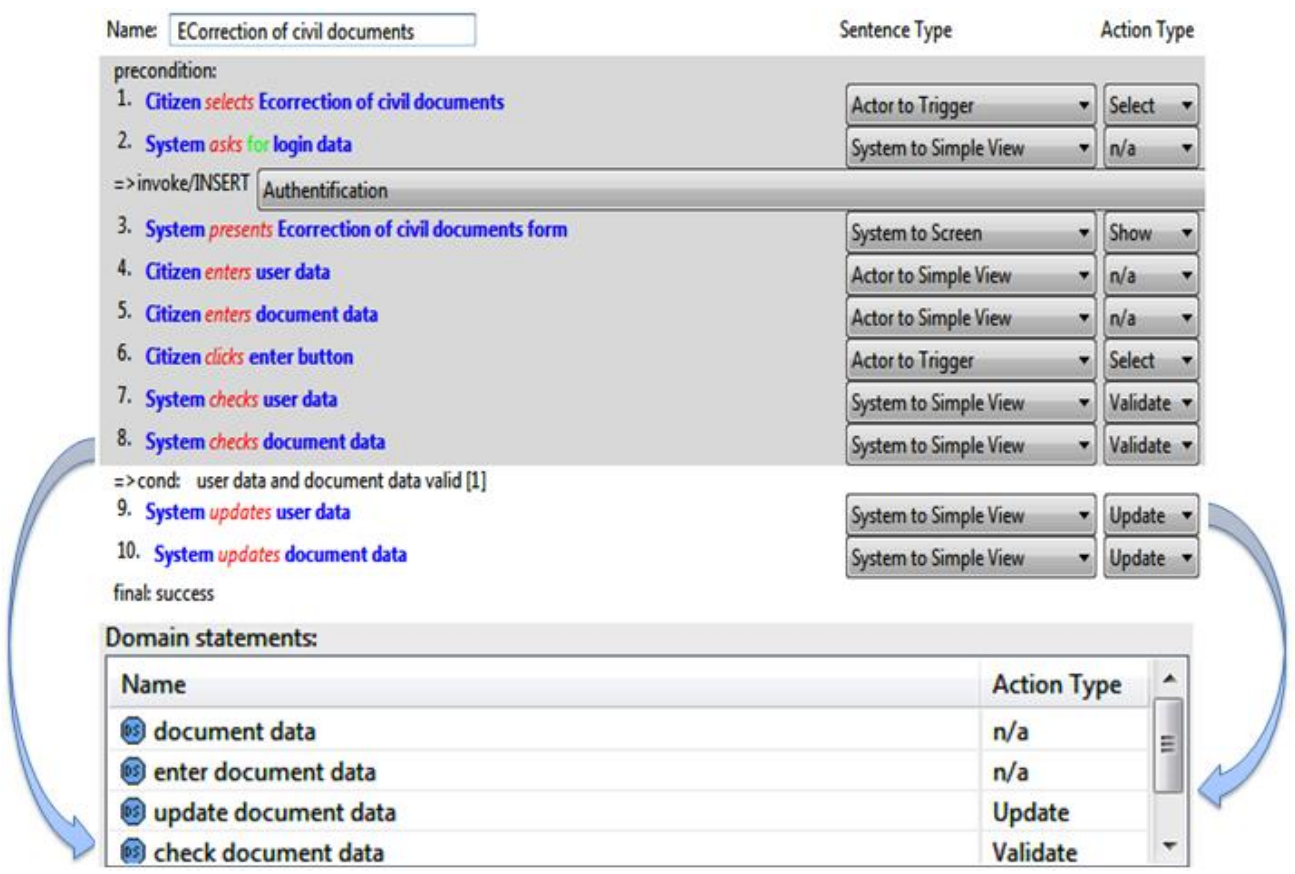


Fig. 4 Abstract syntax for SVO sentences



Name: ECorrection of civil documents

	Sentence Type	Action Type
precondition:		
1. Citizen selects Ecorrection of civil documents	Actor to Trigger	Select
2. System asks for login data	System to Simple View	n/a
=>invoke/INSERT Authentification		
3. System presents Ecorrection of civil documents form	System to Screen	Show
4. Citizen enters user data	Actor to Simple View	n/a
5. Citizen enters document data	Actor to Simple View	n/a
6. Citizen clicks enter button	Actor to Trigger	Select
7. System checks user data	System to Simple View	Validate
8. System checks document data	System to Simple View	Validate
=>cond: user data and document data valid [1]		
9. System updates user data	System to Simple View	Update
10. System updates document data	System to Simple View	Update
final: success		

Domain statements:

Name	Action Type
document data	n/a
enter document data	n/a
update document data	Update
check document data	Validate

Fig. 5 SVO sentences in domain statement

The other actions ("Select" and "Show") are related to the other elements of the domain (triggers and screens). In each sentence of the scenario, the subject points to a "NounPhrase" with different subject types (Actor, System). The "Predicate" points to a "VerbPhrase" which is contained as a name of the "Domain Statement" of a certain notion of domain (here "user data", as shown at the bottom of Fig. 5.

III. Scenario Translational Semantics for Database Access

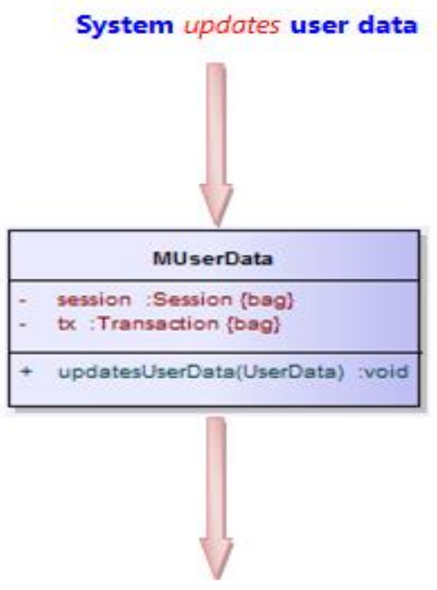
This section describes transformation rules for generating a database access code from use case scenarios to persist and store data in a database system. The semantics of RSL are defined by translating it into another language. This approach is called "Translational semantics". However, RSL has to be considered as a superior language to 3GL (3rd Generation Languages) such as Java to apply this approach. In this case, RSL constructs can be translated into 3GL constructs. The semantics of 3GL are well known. Thus, a transformation program can be built to generate source code from RSL. The source language of the transformation is RSL, and the target language is Java. In this work, Java was chosen because it is widely used by the majority of software developers. The architecture of the target code follows the Model-View-Presenter (MVP) pattern [21]. The View layer represents the GUI elements for displaying data and interacting with the user, and the Presenter layer is in charge of driving the observable behavior of the system, and controlling the sequence of updates made to the Model and View layers. The Model layer is the main focus of this study, as it allows persistence and storage of data in a database. The rules for the other layers were discussed in [6], [12] , and [20].

Next, the transformation rules are cited and, for each rule, an illustration is presented.

- **Rule R1:** Each View-type notion is translated into a Model class. The name of the class is derived from the notion's name, without space, turning it to upper camel case, and adding the prefix "M".
- **Rule R2:** Any notion of view type with associated notions of attribute type is translated into a DTO class. The class name is derived from the notion's name by removing spaces and changing to upper case. The attributes are translated into the attributes of the DTO class. Their names are copied from the names and types of the attribute notions respectively ("Text" to "String", "Whole number" to "Integer", "Real number" to "Float", "True/False" to "Boolean", "Date" to "DateTime"). In addition, each DTO class contains an ID attribute of type "long".
- **Rule R3:** Each "non-read" sentence of the type: "System-to-Simple View" is translated into a class in the Model layer. The sentence must involve an action different from

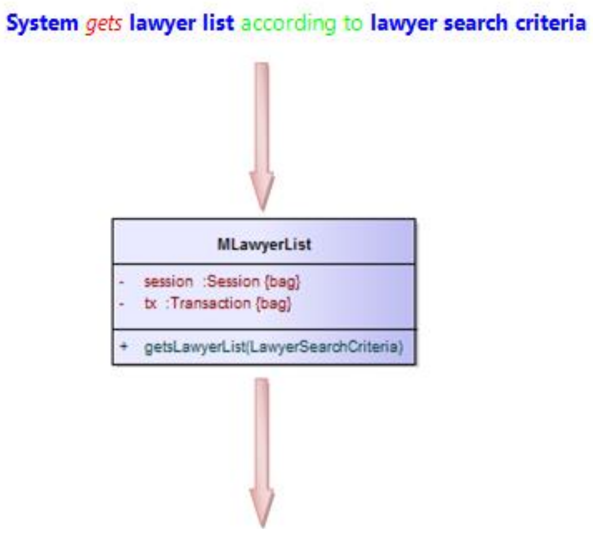
the "Read" action, i.e. "Create", "Update", "Delete", plus "Validate". The name of the operation is derived from the predicate name of the sentence without spaces, and turns it into camel-case format. Fig. 6 illustrates the rule. The "user data" simple view notion in the sentence "System updates user data" is translated into the UML class "MUserData". This class contains the method: "updatesUserData", which is treated as an "UPDATE" action as explained in Section II (Subsection C). Then, the UML class is transformed into a Java class. The lines 15-28 show the "updatesUserData" method which takes the name of the notion as parameter, and updates it using the "update" method of the session object. The "session" object is an instance of the "Session" class in Hibernate, used to obtain a physical connection with a database. It allows to persist and retrieve data each time it is instantiated. It must not be opened for a long period of time, using the "close" method. The transaction object "tx" is also a unit of work with the database. Hibernate's common () method is used to initiate objects in Hibernate and is inherited from a generic model class.

- **Rule R4:** Each "Read" System-to-ListView (Simple View) sentence is translated into a class in the Model layer. The sentence must involve a "Read" action, and may contain one or two objects (direct or both direct and indirect). The direct object points to a List View and the indirect object, if it exists, points to a Simple View. The name of the operation is derived from the concatenated predicate in the direct object. Fig. 7 shows the description of the rule. The notion of "lawyer list" in the sentence "System gets lawyer list according to lawyer search criteria" is translated into the UML class "MLawyerList", with the "getsLawyerList" method. It is then transformed into a Java class (see lines 16-37) and contains the notion "LawyerSearchCriteria" as a parameter, which is the indirect object. It uses Java iteration constructs to loop a list of data and print certain information for the user.
- **Rule R5:** Each "Read" System-to-Simple View (List View) sentence is translated into a class in the Model layer. The sentence must involve a "Read" action, and can have direct and indirect objects. The direct object refers to the Simple View and the indirect object, if it exists, refers to a List View. The operation's name is derived from the predicate to the direct object (Simple View). Fig. 8 shows the rule. The direct object "lawyer data" in the sentence "System gets lawyer data" is similarly translated into the model class "MLawyerData", and contains the "getsLawyerData" method, which is treated as a "Read" action. Then it is translated into a Java class. The method (lines 16 to 29) takes an "ID" identifier as a long type parameter, for example, to identify the data in the database table, and uses the "load" method of the session object, to read the data.



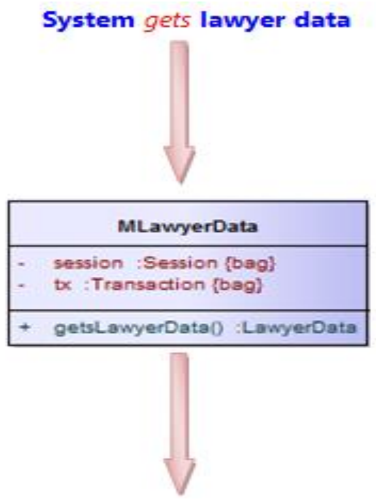
```
1
2 public class MUserData extends
GenericModel {
3
4    private Session session;
5    private Transaction tx;
6
7    public MUserData(){
8
9    }
10
11    public void finalize() throws
Throwable {
12
13    }
14    public void
updatesUserData(UserData userData){
15        try{
16         common();
17         session.update(userData);
18         session.flush();
19          tx.commit();
20        }   catch
(HibernateException e){
21         if (tx!=null)
tx.rollback();
22         e.printStackTrace();
23          }
24          finally{
25          session.close();
26        }
27    }
28 }
```

Fig. 6 Illustration of Rule R3


System gets lawyer list according to lawyer search criteria
MLawyerList
- session :Session {bag}
- tx :Transaction {bag}
+ getsLawyerList(LawyerSearchCriteria)

```


public class MLawyerList extends GenericModel {

   private Session session;
   private Transaction tx;

   public MLawyerList(){

   }

   public void finalize() throws Throwable {

   }

  public List getsLawyerList(LawyerSearchCrietria lawyerSearchCriteria){
       List list= null;
        try{
        common();
        list = session.createQuery("from LawyerSearchCriteria").list();
        Iterator iterator = list.iterator();
        while (iterator.hasNext()) {
        LawyerSearchCriteria lawyerList= (LawyerSearchCriteria) iterator.next();
System.out.println(lawyerList +" ");
       }
      .........
        }
```

Fig. 7 Illustration of Rule R4



```


 public class MLawyerData extends GenericModel {

   private Session session;
   private Transaction tx;

    public MLawyerData(){

    }

    public void finalize() throws Throwable {

    }

   public LawyerData getsLawyerData(long ID){
   LawyerData lawyerddata=null;
       try{
        common();
lawyerData=(LawyerData)session.load(LawyerData.class, ID);
       }  catch (HibernateException e){
       if (tx!=null)
        tx.rollback();
       e.printStackTrace();
        }
        finally{session.close();
       }
    return LawyerData;
    }

    }
```

Fig. 8 Illustration of Rule R5

IV. CODE GENERATION

A. Model Transformation Language:MoLA

The aforementioned translation rules were implemented using the MoLA transformation language. This language was chosen because it can be easily interfaced with RSL's modeling environment: i.e.: editor and model repository. In addition, it can produce models in UML and has good text processing capabilities. MoLA is a procedural graphical language that combines two programming paradigms: declarative and imperative, developed at the University of Latvia, IMCS, and supported by the Eclipse-based METAclipse tool [22] and [23]. In graph transformations, the declarative paradigm is specified by specifying models that must be found or generated in the model graphs. When executing a declarative rule, certain objects in the model graph are found or updated, and are then available for further processing via references. The results of declarative processing can be used by imperative elements, which can define sequences in which the declarative rules are to be executed. The basic element in MoLA is a rule, which consists of a "pattern", and "actions". A pattern is a class element that conforms to a metamodel. In addition to pattern matching, a MoLA rule also performs other tasks: it creates a class instance or link, deletes an instance or link, or changes (modifies) the values of an instance. A MoLA procedure - the executable transformation unit - is built from rules, using constructions from traditional structural programming: loops, branchings, If-then-Else, procedure calls in graphical forms. All these look like UML activity diagrams. It can also contain declarations: parameters and variables (primitives, and class). The next paragraph discusses MoLA constructs by giving examples of transformation procedures that implement the rules described in Section III.

B. Transformation Algorithm

This sub-section presents the implementation of the semantics from RSL to Java presented in Section III. MoLA was used to write the transformation rules. This work focuses on the rules that allow to generate the database access code, and the associated Hibernate code to persist and store the data in a database.

Fig. 9 illustrates the main MoLA procedure, which contains six procedure calls. This main procedure is the implementation of the process shown in

. The first two procedures are used to organize the package structure for the entire application. The third procedure is used to create the hibernate configuration file. This file contains all required database data and other related parameters. The fourth procedure is used to generate the DTOs classes, which are called persistent classes in Hibernate. They must contain an ID to easily identify objects in Hibernate and the database. The ID attribute is mapped to the primary key column in a database table. Other attributes must be declared as private, and have getter/setter methods. These classes are used by Hibernate for mapping, because Hibernate maps DTOs to database tables and database views, and maps Java data types to SQL data types. The fifth procedure generates mapping files for each DTO class, which contain the mapping information defining how the Java classes are linked to the database tables. It also contains information about the associations between tables: One-To-One, One-To-Many, Many-To-One, and Many-To-Many mappings. The last procedure is used to generate the database access code, i.e. the Model layer. This procedure is used to generate classes for manipulating data in a database system using the popular CRUD methods. Fig. 10 illustrates the details of this procedure. It iterates on each notion of an RSL model, whose type can be either a SimpleView ("tagNonpersistent") or a List View ("tagList"), and creates a class with the prefix "M" as indicated in the above rules concatenated with the notion name (notion's name) in the "Model" package. Then, it loops SVO sentences that are linked to a certain domain statement, and creates the necessary dependencies. The remaining instructions allow to get the action using the verb name of the sentence. Then, the appropriate method will be called to create, read, update or delete the data from the database, following the rules described in Section III.

Fig. 11 shows the implementation of rule R3. This procedure takes four input parameters, and creates an operation (op: Operation) for a Model class (@cl:Class). The name of the operation takes the name of the verb concatenated with the notion's name. And the parameter's name is the name of the DTO class in camel-case. And, then the code of the operation is inserted according to the type of action ("Create", "Update", "Delete" or "Validate"). The Hibernate's predefined operations that are linked to the Session class (for example, session.save(object), session.update(object), and session.delete(object)). For the "Validate" action, only the code skeleton is generated. Rule R4 (Fig. 12), allows reading data from a database table. This procedure takes three input parameters and, like the previous procedure, it creates an operation for the model class and inserts the code of the operation which will be repeated on the objects of the table/view. Then, it prints the data required by the user (for example, ID, name, or other data). Fig. 12 shows that the operation takes a parameter which is the DTO object in this case, because there is an indirect object in the SVO sentence. Otherwise, if there is no indirect object, the method will have no parameters.

V.CASE STUDY

A. Validation Approach

The proposed approach was applied in the "Tribunal E-Services" system to test its reliability. This is the domain problem defined using the RSL's domain model notation. The example contains user-system interactions, which are defined using scenarios. The system uses a dozen interconnected use cases of four types related to citizens, lawyers, documents

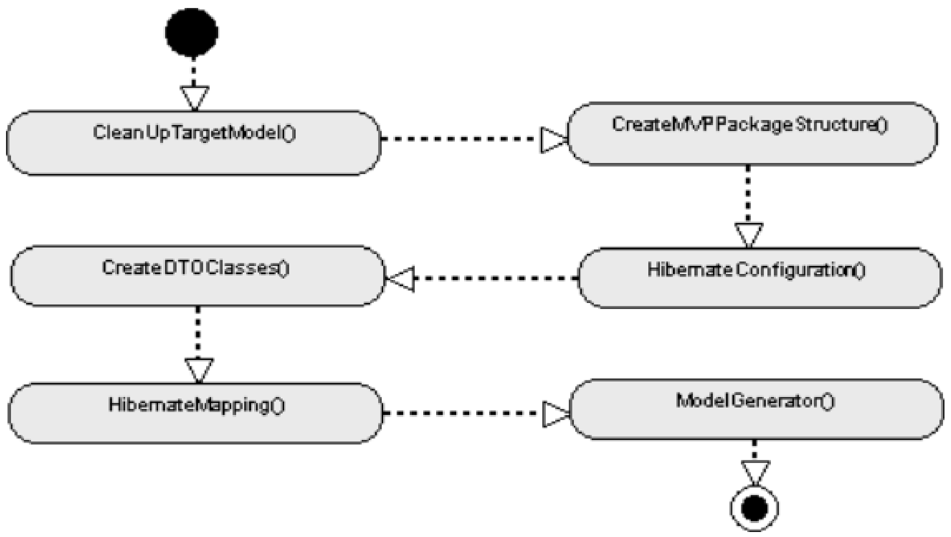


Fig. 9 Main MoLA Procedure

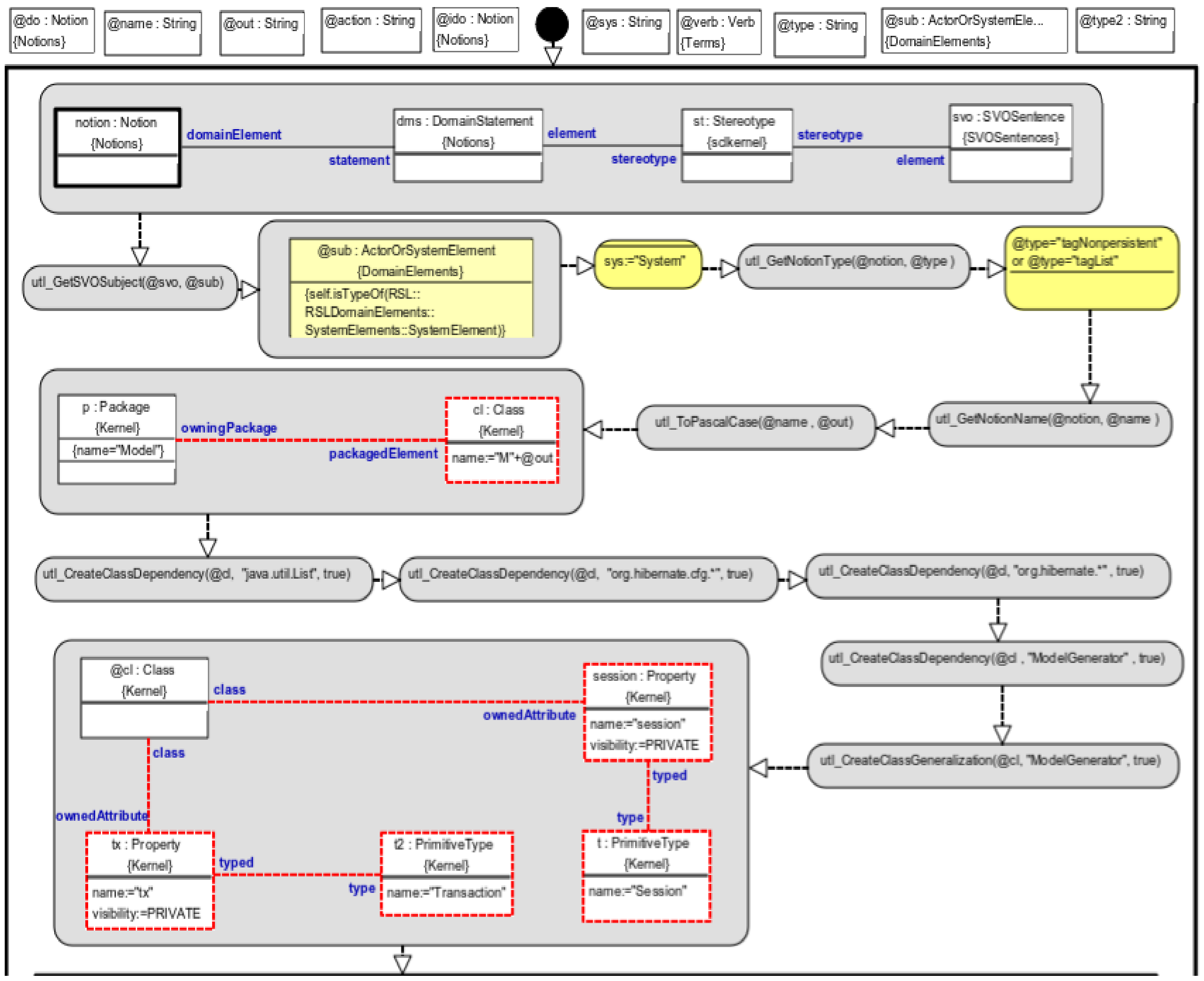

@do : Notion {Notions}
@name : String
@out : String
@action : String
@ido : Notion {Notions}
@sys : String
@verb : Verb {Terms}
@type : String
@sub : ActorOrSystemEle... {DomainElements}
@type2 : String
notion : Notion {Notions}
domainElement
statement
dms : DomainStatement {Notions}
element
stereotype
st : Stereotype {sclkernel}
stereotype
element
svo : SVOSentence {SVOSentences}
utl_GetSVOSubject(@svo, @sub)
@sub : ActorOrSystemElement {DomainElements}
{self.isTypeOf(RSL::RSLDomainElements::SystemElements::SystemElement)}
sys:="System"
utl_GetNotionType(@notion, @type)
@type="tagNonpersistent" or @type="tagList"
p : Package {Kernel}
{name="Model"}
owningPackage
packagedElement
cl : Class {Kernel}
name:="M"+@out
utl_ToPascalCase(@name , @out)
utl_GetNotionName(@notion, @name)
utl_CreateClassDependency(@cl, "java.util.List", true)
utl_CreateClassDependency(@cl, "org.hibernate.cfg.*", true)
utl_CreateClassDependency(@cl, "org.hibernate.*" , true)
utl_CreateClassDependency(@cl , "ModelGenerator" , true)
utl_CreateClassGeneralization(@cl, "ModelGenerator", true)
@cl : Class {Kernel}
class
ownedAttribute
session : Property {Kernel}
name:="session"
visibility:=PRIVATE
typed
type
class
ownedAttribute
tx : Property {Kernel}
name:="tx"
visibility:=PRIVATE
typed
type
t2 : PrimitiveType {Kernel}
name:="Transaction"
t : PrimitiveType {Kernel}
name:="Session"

Fig. 10 MoLA procedure for the Model generator (a)

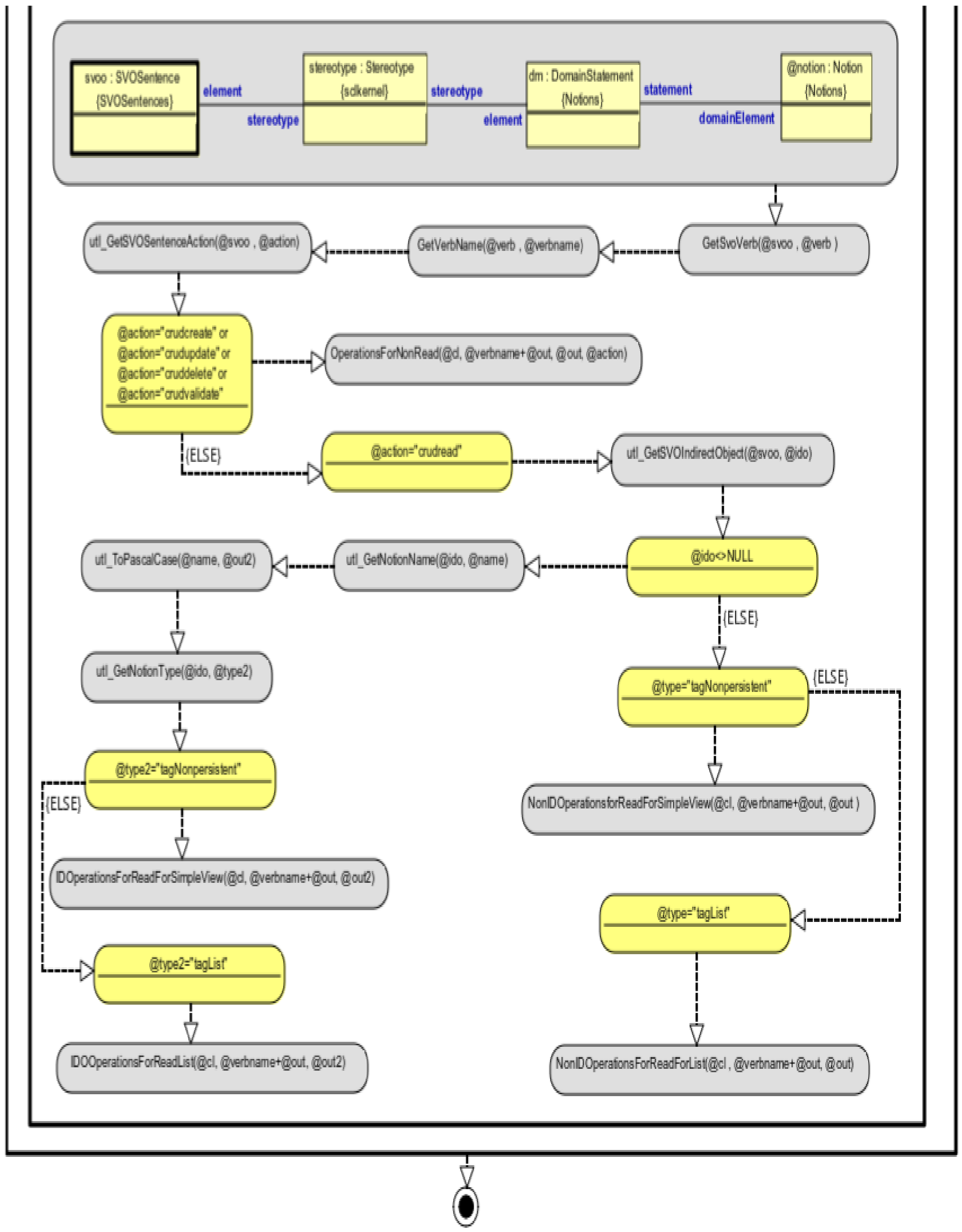

svoo : SVOSentence {SVOSentences}
element
stereotype
stereotype : Stereotype {sdkernel}
stereotype
element
dm : DomainStatement {Notions}
statement
domainElement
@notion : Notion {Notions}
utl_GetSVOSentenceAction(@svoo , @action)
GetVerbName(@verb , @verbname)
GetSvoVerb(@svoo , @verb)
@action="crudcreate" or @action="crudupdate" or @action="cruddelete" or @action="crudvalidate"
OperationsForNonRead(@cl, @verbname+@out, @out, @action)
{ELSE}
@action="crudread"
utl_GetSVOIndirectObject(@svoo, @ido)
utl_ToPascalCase(@name, @out2)
utl_GetNotionName(@ido, @name)
@ido<>NULL
{ELSE}
utl_GetNotionType(@ido, @type2)
@type="tagNonpersistent"
{ELSE}
@type2="tagNonpersistent"
{ELSE}
NonIDOperationsforReadForSimpleView(@cl, @verbname+@out, @out)
IDOperationsForReadForSimpleView(@cl, @verbname+@out, @out2)
@type="tagList"
@type2="tagList"
IDOOperationsForReadList(@cl, @verbname+@out, @out2)
NonIDOperationsForReadForList(@cl , @verbname+@out, @out)

Fig. 10 MoLA procedure for the Model generator (b)

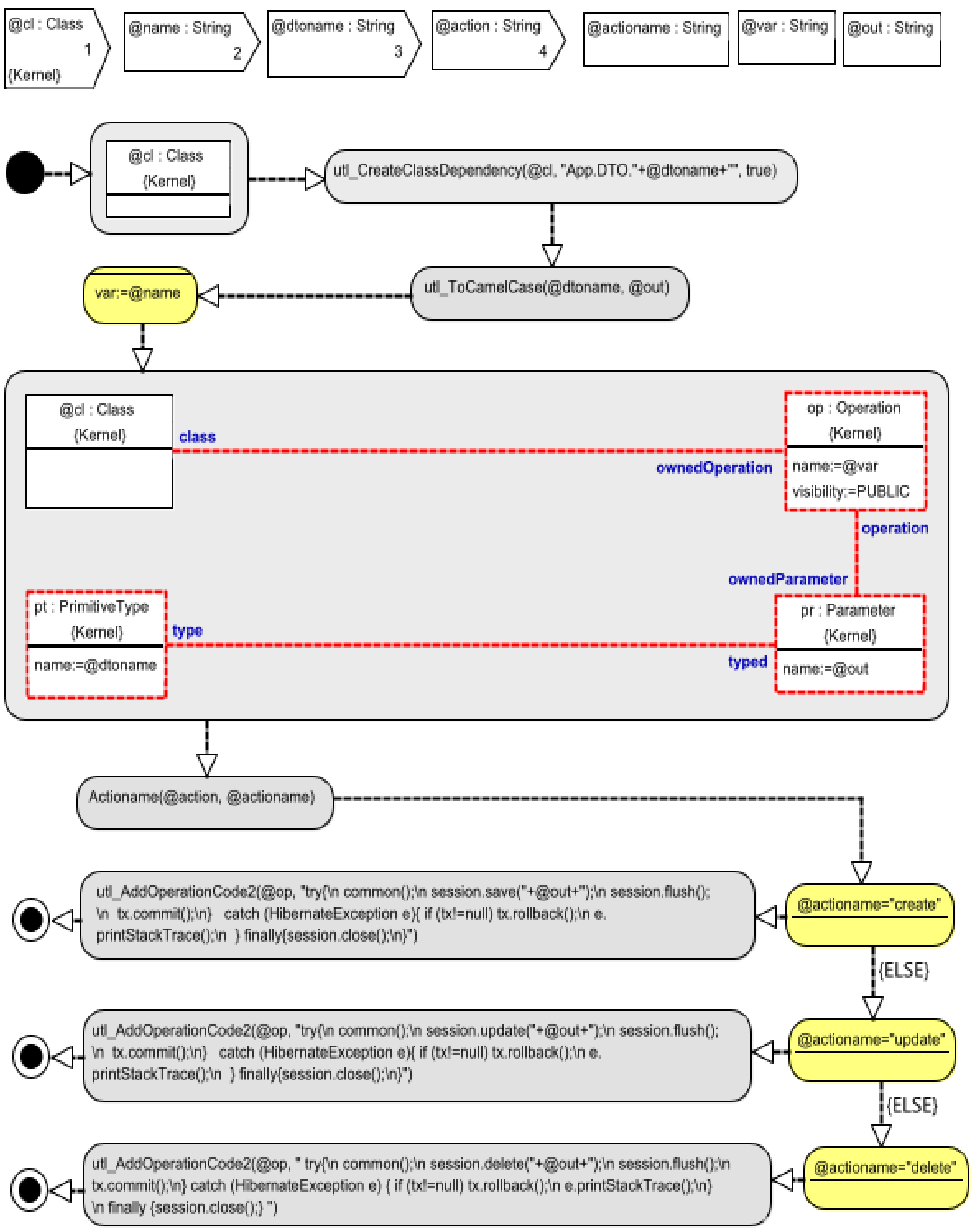
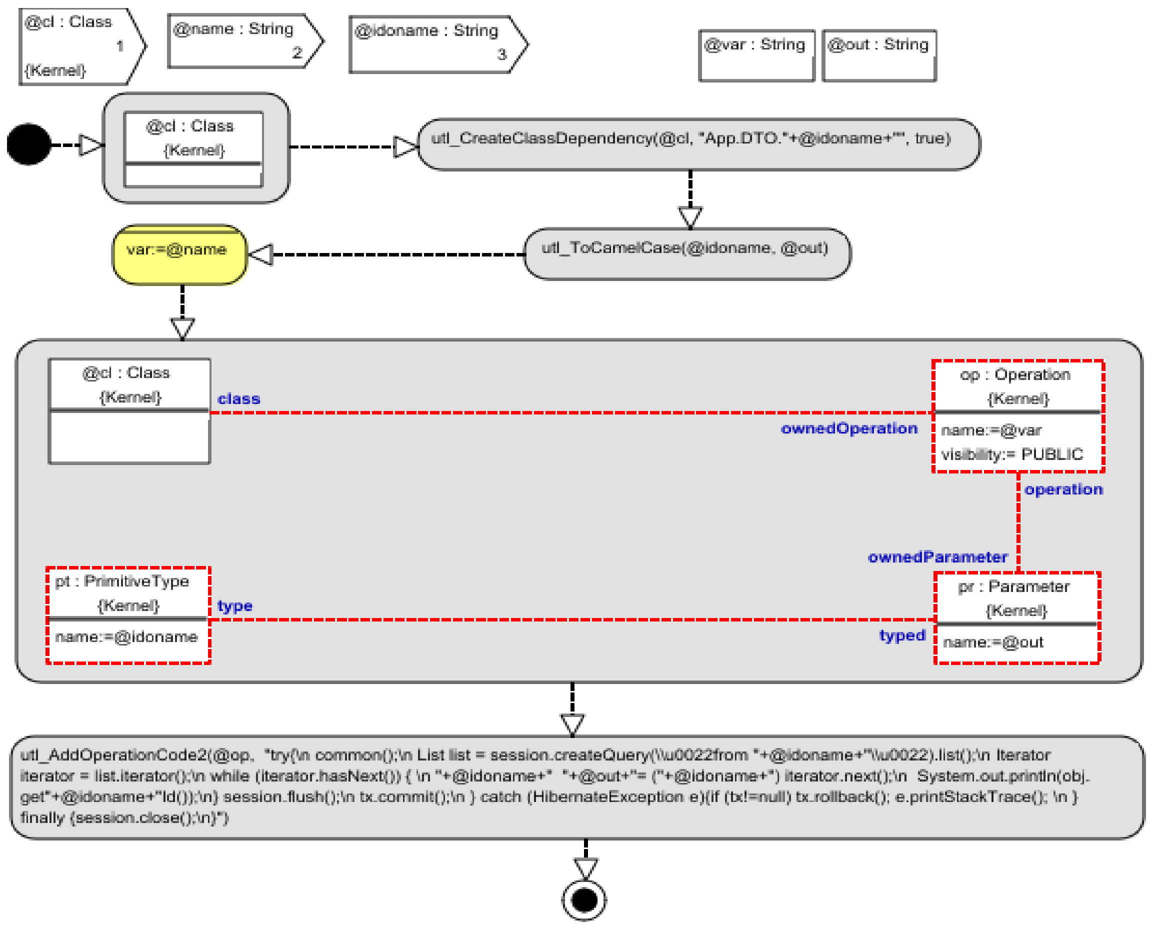

@cl : Class 1 (Kernel)
@name : String 2
@dtoname : String 3
@action : String 4
@actionname : String
@var : String
@out : String
@cl : Class (Kernel)
utl_CreateClassDependency(@cl, "App.DTO."+@dtoname+"", true)
utl_ToCamelCase(@dtoname, @out)
var:=@name
@cl : Class (Kernel)
class
ownedOperation
op : Operation (Kernel)
name:=@var
visibility:=PUBLIC
operation
ownedParameter
pt : PrimitiveType (Kernel)
name:=@dtoname
type
typed
pr : Parameter (Kernel)
name:=@out
Actionname(@action, @actionname)
utl_AddOperationCode2(@op, "try{\n common();\n session.save("+@out+");\n session.flush();\n tx.commit();\n} catch (HibernateException e){ if (tx!=null) tx.rollback();\n e.printStackTrace();\n } finally{session.close();\n}")
@actionname="create"
{ELSE}
utl_AddOperationCode2(@op, "try{\n common();\n session.update("+@out+");\n session.flush();\n tx.commit();\n} catch (HibernateException e){ if (tx!=null) tx.rollback();\n e.printStackTrace();\n } finally{session.close();\n}")
@actionname="update"
{ELSE}
utl_AddOperationCode2(@op, " try{\n common();\n session.delete("+@out+");\n session.flush();\n tx.commit();\n} catch (HibernateException e) { if (tx!=null) tx.rollback();\n e.printStackTrace();\n} \n finally {session.close();} ")
@actionname="delete"

Fig. 11 MoLA procedure for Rule R3



```
utl_AddOperationCode2(@op,  "try{\n common();\n List list = session.createQuery(\\u0022from "+@idoname+"\u0022).list();\n Iterator
iterator = list.iterator();\n while (iterator.hasNext()) { \n "+@idoname+"  "+@out+"= ("+@idoname+") iterator.next();\n  System.out.println(obj.
get"+@idoname+"Id());\n} session.flush();\n tx.commit();\n } catch (HibernateException e){if (tx!=null) tx.rollback(); e.printStackTrace(); \n }
finally {session.close();\n}")
```

Fig. 12 MoLA procedure for Rule R4

(criminal record extract, etc.) and other services and other services (e.g. follow judicial affair). The source domain model was developed using the ReDSeeDS (Requirements-Driven Software Development System) framework [14], and the transformation rules using MoLA. Java code was generated using the Enterprise Architect tool generator [24]. Following, snapshots of the case study are presented.

A. Source Model

The overall scope of the system is defined using the conceptual domain model, in addition to use cases, and, must be extended by further details of the application logic.

The RSL source model includes eleven use cases, as shown in Fig. 13. Four of the use cases are connected to the actor by a "use" relationship. The remaining use cases are interconnected by the "invoke" relationship. The problem domain illustrated in Fig 14 comprises seven concepts related to the citizen and different e-services that can be realized with this system. Of course, these concepts have attributes and are linked by associations that have multiplicities (1, *) as explained in Section II (Subsection B).

Fig. 14 shows a part of the RSL model as a whole. One use case, "Extract nationality", represented by a scenario in Fig. 15, is described. The scenario begins with an "Actor-to-Trigger" sentence that defines the initial user interaction.

Another relationship between "Data Views" and "Concepts" in the RSL can also be distinguished, called "main concept
Fig. 15 shows a relationship between the "user data" simple view and the "citizen" concept. Sentence 3 is a "System-to-Screen" sentence to show a screen element (here: "System shows nationality form"). The following sentences allow the actor to edit (sentence 4) the data. Then the system checks the data, if it is correct, then the system saves it (sentence 7), and the main scenario ends with two SVO phrases, which allow the user to view the data and close the form. If not, it displays a message (quell message to indicate error), and asks the user to fill in the data again. If this is the case, the main scenario is rejoined from the sentence "Citizen fills user data", using the "rejoin" relationship. Otherwise, the system will close the form. The remaining use cases and their stories are similar to the "Extract nationality" scenario, and they are not presented in this study.

A. Target Code

Once the implementation of the RSL source model and all rules is complete, the transformation can be executed (main procedure in Fig. 9). The target model includes Java code. The proposed code can be easily combined with the code generated in the works of Michał Śmiałek and Wiktor Nowakowski [6], and [12] in which the researchers have generated code following the Model-View-Presenter design pattern.

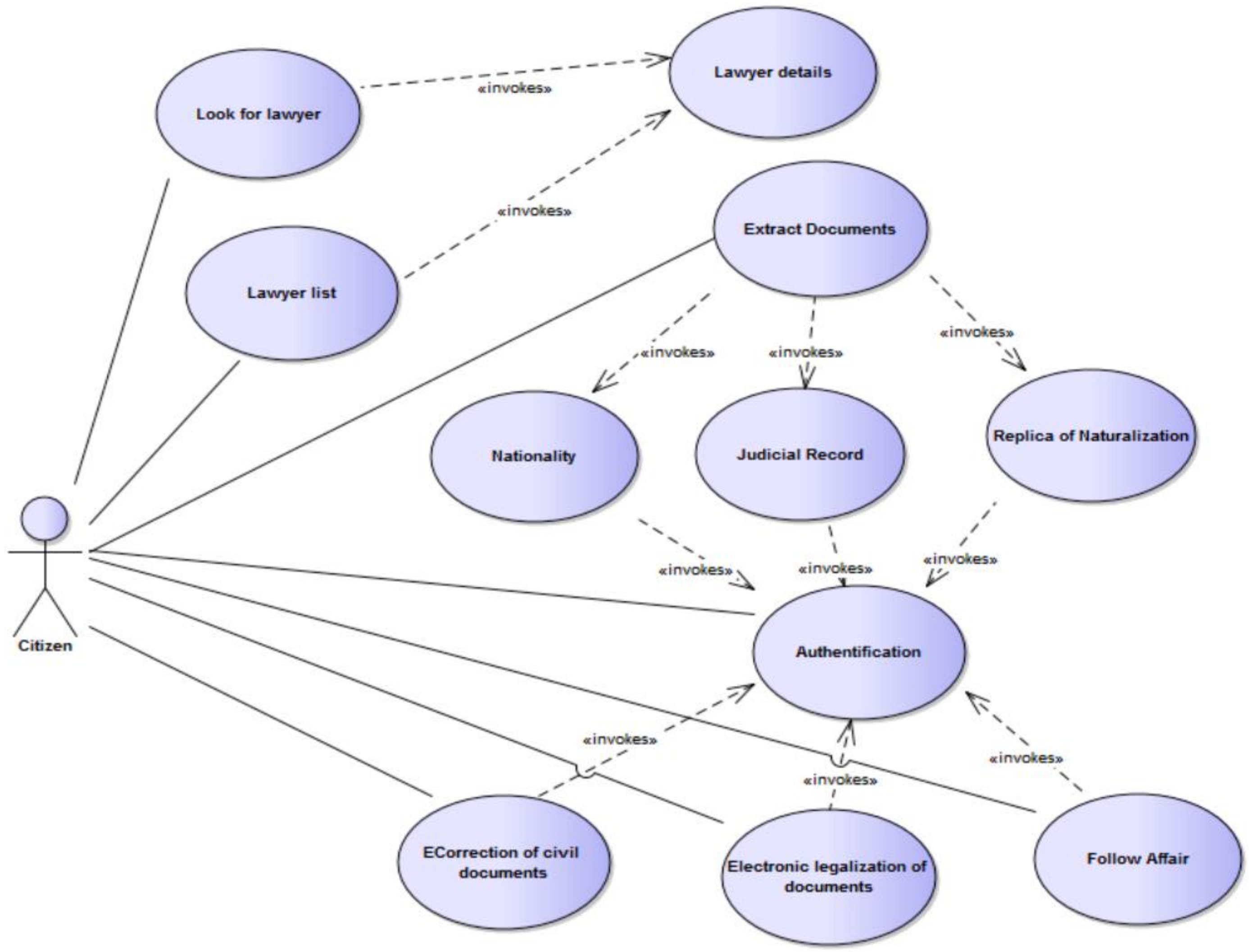


Fig. 13 Use case diagram for "Tribunal E-Services" system

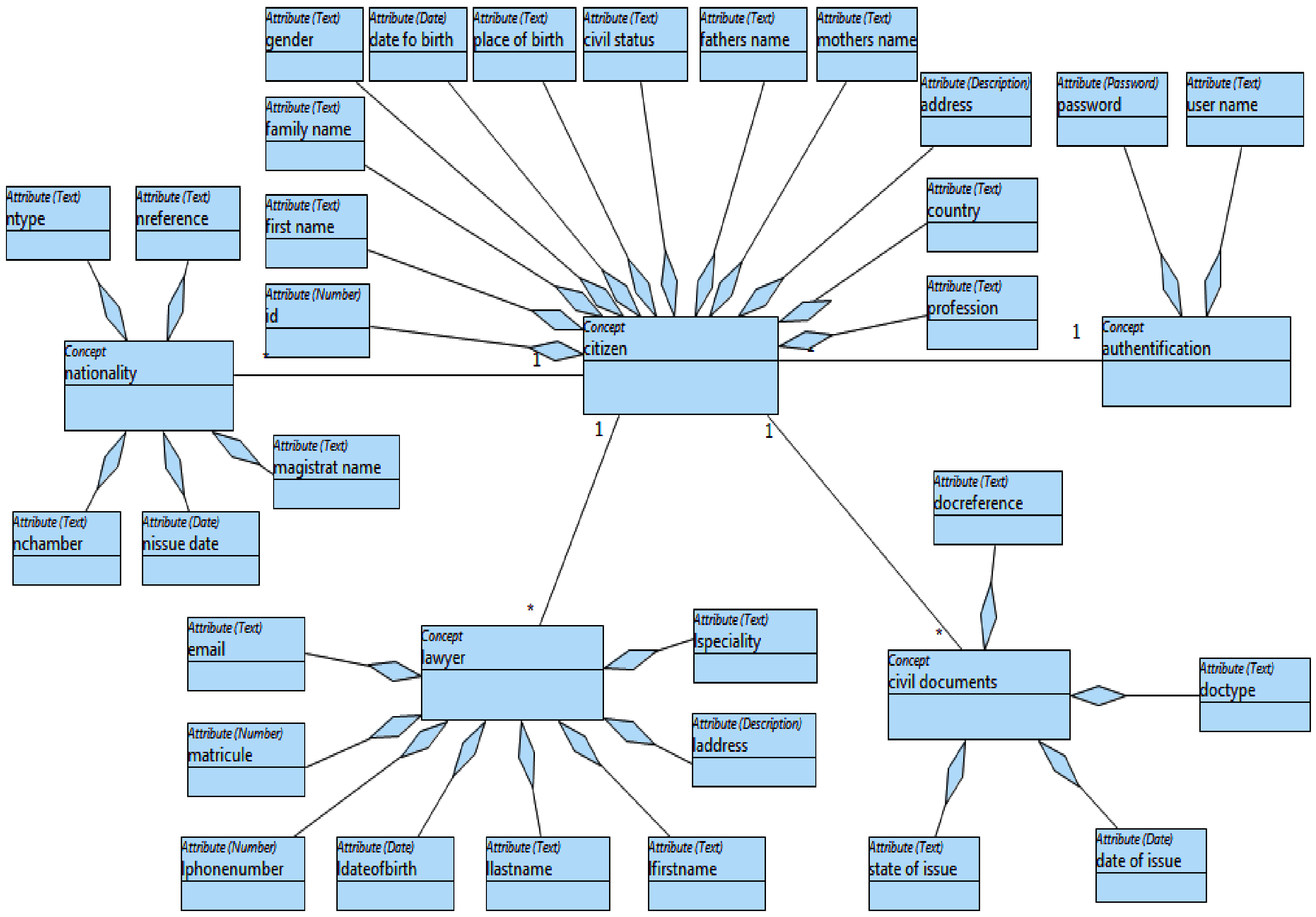
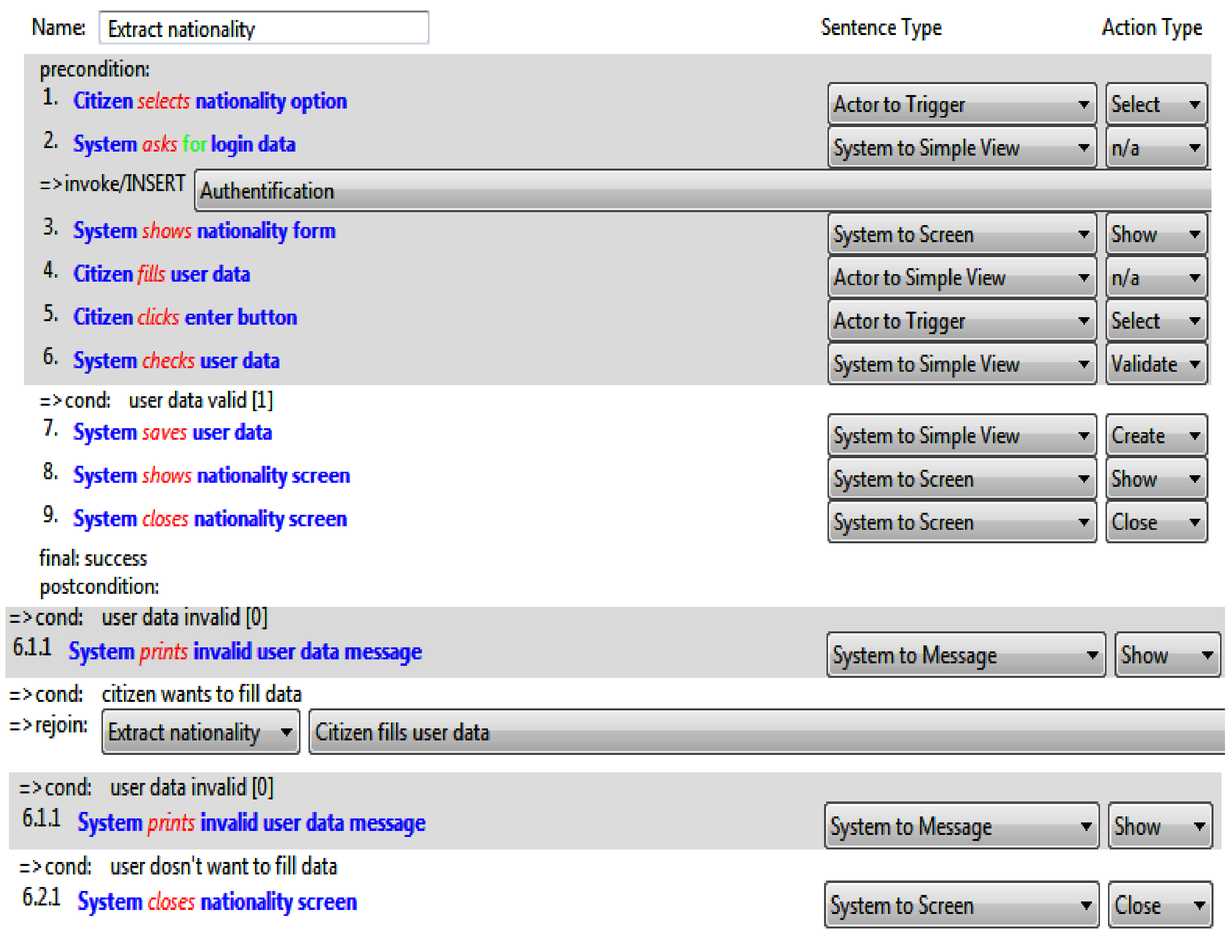

Attribute (Text) gender
Attribute (Date) date fo birth
Attribute (Text) place of birth
Attribute (Text) civil status
Attribute (Text) fathers name
Attribute (Text) mothers name
Attribute (Description) address
Attribute (Password) password
Attribute (Text) user name
Attribute (Text) family name
Attribute (Text) ntype
Attribute (Text) nreference
Attribute (Text) first name
Attribute (Text) country
Attribute (Number) id
Attribute (Text) profession
Concept nationality
Concept citizen
Concept authentification
1
1
1
1
Attribute (Text) magistrat name
Attribute (Text) docreference
Attribute (Text) nchamber
Attribute (Date) nissue date
*
*
Attribute (Text) email
Concept lawyer
Attribute (Text) lspeciality
Concept civil documents
Attribute (Text) doctype
Attribute (Number) matricule
Attribute (Description) laddress
Attribute (Number) lphonenumber
Attribute (Date) ldateofbirth
Attribute (Text) llastname
Attribute (Text) lfirstname
Attribute (Text) state of issue
Attribute (Date) date of issue

Fig. 14 Part of the RSL domain model



Name: Extract nationality

	Sentence Type	Action Type
precondition:		
1. Citizen *selects* nationality option	Actor to Trigger	Select
2. System *asks* for login data	System to Simple View	n/a
=>invoke/INSERT Authentification		
3. System *shows* nationality form	System to Screen	Show
4. Citizen *fills* user data	Actor to Simple View	n/a
5. Citizen *clicks* enter button	Actor to Trigger	Select
6. System *checks* user data	System to Simple View	Validate
=>cond: user data valid [1]		
7. System *saves* user data	System to Simple View	Create
8. System *shows* nationality screen	System to Screen	Show
9. System *closes* nationality screen	System to Screen	Close
final: success		
postcondition:		
=>cond: user data invalid [0]		
6.1.1 System *prints* invalid user data message	System to Message	Show
=>cond: citizen wants to fill data		
=>rejoin: Extract nationality / Citizen fills user data		
=>cond: user data invalid [0]		
6.1.1 System *prints* invalid user data message	System to Message	Show
=>cond: user dosn't want to fill data		
6.2.1 System *closes* nationality screen	System to Screen	Close

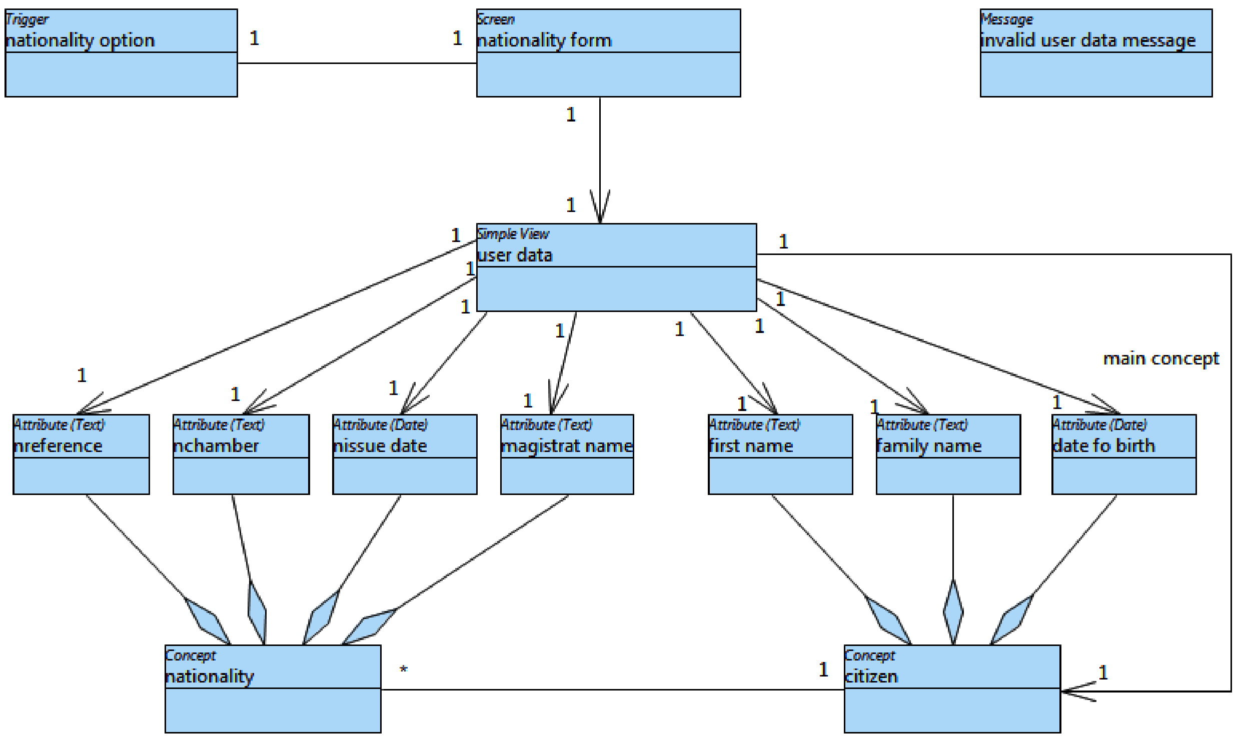


Fig. 15 Details for "Extract nationality" use case

They generated a code for the Presenter and View layers, and only a code skeleton for the Model layer for both Desktop and Web-based applications [12]. In the present work, the code of the Model layer is generated to persist and store the data in a database system using CRUD methods and Hibernate ORM for a Desktop-based application.

Fig. 16 shows the structure of the translational framework. It can be seen that the classes of the three layers are linked by associations; the associations between the Presenter layer and the View layer are bidirectional, and the associations between the Presenter layer and the Model layer are unidirectional. In this MVP variant, there is no association between the View and Model layers. By convention, View class names begin with the letter "V", Presenter class names begin with "P" and Model class names begin with "M", as mentioned above.

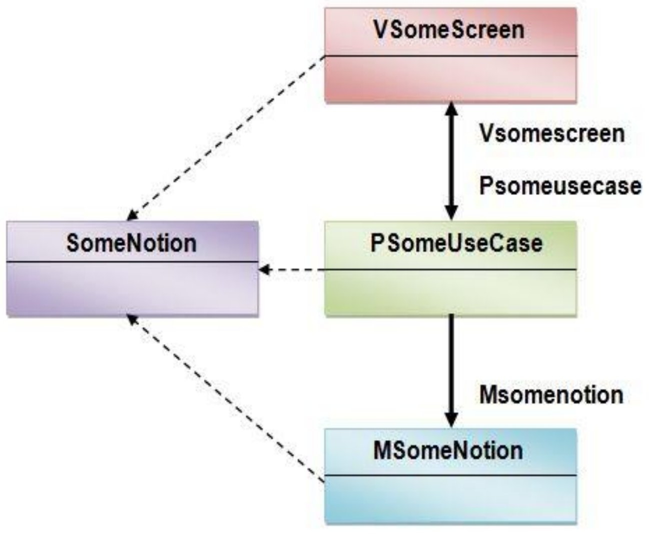


Fig. 16 Structure of the translational framework

VI. Related Work and comparison

In this section, we survey earlier works that allow to derive target models (structural or behavioral) from requirements. After that, we discuss approaches that are able to generate source code from requirements. At the end of this section, we compare our approach with recent approaches (last five years) in this area.

Ding et al. [25] proposed a method in which they automatically adapted the Cockburn's use cases to IBM and Microsoft Service Component Architecture (SCA) models, using the Taurus tool and the Atlas Transformation Language (ATL) [26] to write the transformation rules, in addition to the Eclipse ATL Development Tools (ADT) platform.

Santonu Sarkar et al. [27] proposed a solution called "Design Assistant Tool" (DAT) to create a detailed design (based on UML) from requirements and use case texts written with natural language processing (NLP). This tool combines natural language processing techniques and design heuristics.

Yue et al. [28] proposed a method and a framework called aToucan that allows to automatically derive UML analysis models (class, sequence, activity diagrams) from documented functional requirements such as UCMod (use case diagrams, natural language information, etc.). The authors presented a MOF-based use case metamodel; UCMeta models use case models that follow RUCM.

Recent pertinent works that can generate executable code automatically from requirements models are cited below.

The authors in [29], proposed an approach for generating Java code automatically. They extract a semantic model representation from requirements written in any natural language (English, French … etc.), and then covert this model into a code called: Pegasus_code, and finally, they refine and transform the Pegasus_Code into Java code. To support their approach, they implemented a tool called: Code Recovery tool (CodeRec-tool). To proof the feasibility of their approach, they presented a case study for "library management system".

The approaches cited in the next paragraphs apply the MDD/MDA principles to produce code for Web-based applications. There are also approaches that can generate code for Rich Internet Applications (RIA), which are not discussed here.

Christoforos Zolotas et al. [30] presented an MDE engine for generating RESTful web services from textual requirements (based on SVO motif) and their visual storyboards, following the MDA technology [31]. For this, they developed two modules as Eclipse plung-ins that are: Reqs2Specs modules and an MDE Engine. Reqs2Specs was used to write the requirements (with: requirements editor, and Storyboard creator). The MDE engine was used to apply transformations from CIM to PIM, and from PIM to PSM with ATL language, and from PSM to code with Acceleo [32]. Their mechanism covers non-CRUD functionalities such as: Authentication, keyword searching, and also: interoperability with existing 3rd party services. To examine the validity of their approach, the authors provided the RestMarks case study.

Imane Essebaa et al. [33] introduced a tool support to automate the MDA process for MVC Web Application called: "MoDAr-WA". To model their requirements, the authors used business vocabulary and business rules of the SBVR OMG standard [34] at the Computational Independent Level (CIM) level, and then, they generate a use case diagram from SBVR at the same level. After that, they generate the PIM level (class and sequence diagrams) from CIM. The PSM level is generated from PIM level, for web application (detailed class and detailed sequence diagrams). The transition between these MDA models was performed with QVT-Operational language [35]. At the end of the process, the generation of code from PSM is possible with Acceleo tool. The design of the different MDA models was done with Papyrus modeling environment [36]. The MoDAr-WA tool was implemented as an Eclipse plug-in. The validity of this approach has been tested for a Music Store System case study.

Another prominent work was given by [37]. This approach is very similar to [33]. The authors also focus on generating source code automatically for web application from CIM models down to source code following the MVC three-tiered architecture. To define requirements, at the CIM level, the authors propose to deploy use cases: Business Actors, Business UC Realization Diagram, the Business Object Diagrams, and the Sequence Diagrams of the Business scenarios. The transitions from CIM models PIM and PSM models, and finally to code are conducted by a proprietary tool called: xGenerator. The researchers used StarUML [38] to design the UML models at the different MDA layers. They presented An Automated Teller Machine (ATM) as a case study.

The Use case Specification Language (USL) is a concurrent DSL language for specifying precise use cases introduced in [39], The aim is to open the possibility to transform the USL models into other software artifacts.

There are some model-driven tools that can generate an implementation from design models. The Procasor tool [40] can generate source code from use cases and transform them into behavior protocols (procases) and UML state machine diagrams; then, a generator takes these procases and the domain model (UML class diagram), and generates an executable implementation. In this approach, the implementation is composed of three layers: a Presentation layer (pages), a middle layer (business) and a Data layer (generation of classes from UML class diagrams). The JBehave tool [41] is a concurrent tool that can generate source code from user stories. AndroMDA [42] can generate source code from UML models. eMOFLON [43] also generates Java code from UML class diagrams using Enterprise Architect.

These presented approaches can produce code or models (structural or behavioral) from high-level requirements formulated as paragraphs of text (NLP based- approaches…), or use case diagrams and their scenarios (textual or visual) using general purpose modeling languages such as UML, or domain specific modeling languages such as RSL. The authors implemented tools to support the transformation process from requirements to other models or code.

A comparison study between our proposed approach and a list of approaches cited aforementioned (last 5 years) was performed to summarize this section, according to the criteria of different (see Table. I). We focus in this comparison on generating target source code from requirements never less their notation: textual or visual.

The common points between our approach and the compared approaches, are (1) the generation of code from high-level requirements. (2) the utilization of models and model transformations with MDD/MDA-based approaches. (3) the utilization of the Model-View-Presenter/Controller architectural pattern to structure the generated code for our approach and two other approaches.

The main difference between our approach, and the other approaches is the consideration of the visual notation (in our approach) for use cases and their textual scenarios formulated using the domain specific language: RSL under the ReDSeeDS tool.

We concentrate in our work, on generating code for database access code (model layer) directly from use cases (verb sentences) directly with a simple click

Besides the automatic generation of database access code, our approach has also other advantages. The requirements specification language: RSL is easy to understand, and to use. It does not need a high-level background for developers, or domain experts ...etc to be familiar with. From the other side, the ReDSeeDS tool that supports RSL has the possibility to be interfaced with another code generator Modelio [44] – actually-, in addition to the Enterprise Architect tool. Theses modeling tools can visualize and generate code from exported UML models with ReDSeeDS.

To create a ready-to-run web-based application, the transformation engineer can develop some transformation rules easily, and run the program to see the result.

Table. I shows the comparison. The columns present the approaches that are indicated by the authors' names. The rows of the table are the different criteria, which are: source model, target model, requirements tool support, model-to-model transformations, model-to-text transformations, automatic/semi-automatic code generation, generation of database access code, web-based application, and the approach if text in academia or industry.

- **Source model:** the source model in model-driven requirements engineering are: the requirements defined as paragraphs of text and/or accompanied with their use case diagrams. In this study, the languages used for modeling requirements are: UML use case diagrams, OMG's SBVR standard (Semantic of Business Vocabulary And Business Rules), Natural Language Processing (NLP), and Domain Specific Languages (DSLs) as in the proposed approach: the RSL language.
- **Target model:** we mean by the term: target model, the final result of all the MDD/MDA transformation process (M2T transformation) (not the intermediary generated models: such as UML class diagrams and so on). In model-driven approaches, the source code is also treated as a model. The latter can be written in any high-level programming language. As can be seen in Table. I**,** Java language was the most used in this comparison
- **Requirements tool support:** some researchers listed in Table. I developed advanced tools to define requirements models, such as: ReDSeeDS (Requirements-Driven Software Development System) in our approach, Code Recovery tool ([29]), Reqs2Specs ([30]). The others employed existing tools: Papyrus (in [33]), and StarUML (in [37]).
- **Model-to-Model Transformation language:** in M2M transformation, the languages can perform the mapping from source models to other target models (e.g.: UML models). We distinguish text-based and graphical-based transformation languages. Text-based languages used in this comparison are: QVT-O language

Table. I Comparison between model-based code generation approaches from high-level requirements

Criteria \ Approach	**Zolotas et al. [30]**	**Mariem Mefteh et al. [29]**	**Imane Essebaa et al. [33]**	**Gaetanino Paolone et al. [37]**	**Nassima Yamouni-Khelifi et al.**
Source model	Textual scenarios and graphical storyboards	Textual requirements in natural language	Business vocabular and business rules, and UML Use cases diagram	UML Use cases:	Use case diagrams and textual scenarios in RSL Language
Target model	RESTful-web services	Java Code	Java code for Web application	Java code for web application	Java Code
Requirements tool support	Reqs2Specs Eclipse module: Requirements Editor and the Storyboard Creator	Coode Recovery tool	Papyrus Modeling tool (Eclipse Plugin)	StarUML	ReDSeeDS tool
Model-to-Model Transformation language	ATLAS Transformation Language	Transformation rules embedded in the tool	QVT-Operational Language	Transformation rules embedded in the tool: xGenerator	MoLA Language
Model-to-Text Transformation tool	Acceleo	Pegasus	Acceleo	xGenerator	Enterprise Architecture
Automatic/Semi-Automatic of code generation	Automatic	Automatic	Automatic	Automatic	Automatic
Generation of Database access code	Yes (Database schema for authentication and keyword searching)	No	No	No	Yes
Web-based application	Yes	No	Yes	Yes	Available for View and presenter layers
Academic/Industrial case study	Academic	Academic	Academic	Academic/Industrial	Academic

Query/Views/Transformation-Operational), and ATL (Atlas Transformation Language). MoLA is the graphical-based used in our approach. For the rest of the approaches, the transformation rules are embedded in the transformation tool, and the analysts and the designers do not need to care about them.

- **Model-to-Text Transformation tool**: M2T tools can perform the mapping from models to code. As we can seen from Table. I, different tools were used, such as: Acceleo ([30], [33]), Pegasus([29]), xGenerator ([37]), and Enterprise Architect (in our approach).
- **Automatic/Semi-Automatic code generation:** the aim of model-driven requirements engineering is to consider requirements as first-class citizen, in order to contribute in the final code from. The process can be automatic or semi-automatic. In Table. I, we observe that all the approaches, can produce source code automatically from requirements.
- **Generation of database–access code:** most approaches available in the literature focus on the generation of source code in general, but they do not focus on generating database access code to store and maintain data in database systems. The approach of Zolotas et al. [30] can generate code for authentication, and keyword searching . Just, our

approach implements rules for generating CRUD methods automatically.

- **Web-based application:** web-based applications are a trend in the field of software applications because of because of their flexibility and ease of deployment. Most approaches focus on web-code. Our approach will be enhanced to generate code for web-based applications. Meriam et al [29] has also the same issue.
- **Tested in academia or industry**: the researchers listed in Table. I conducted case studies to examine the efficiency and the validity of their proposed methods/tools. The most approaches involved in this comparison were tested in academia. The approach of [37] was adopted by an industry.

VII. VALIDITY LIMITS AND THREATS

While the proposed approach has advantages with respect to automatic code generation from high-level requirement models, it also has some disadvantages and limitations. The main threat to external validity concerns the experimentation in this work. Can the experimental material be used for industrial practice? With regard to this question, since its development in 2009, the ReDSeeDS/RSL platform has proven itself efficiency towards model transformation and code generation, especially with large use case models and their text scenarios. Several case studies have been carried out for both academia and industry, and numerous performance tests have been presented to validate the approach. The product code was related to: DTO classes, and some DAO classes, as well as the Presenter and View classes. Furthermore, this work allows to produce code for database access following the MVC design model for Desktop application actually. The proposed approach was tested for an academic case study (Tribunal E-Services System) with a dozen of use case models and their scenarios, which is not sufficient to prove the accuracy and completeness of the method in the industrial world, where complex models and very large database systems are required. Therefore, this approach needs to be further improved and a more rigorous evaluation of the generated code is required to examine its validity.

Two drawbacks are associated with the generated code:
1) The type of software applications: are they desktop or web-based?
2) Which query language is best suited to be used with Hibernate ORM?
Regarding question 1, the proposed code runs actually on desktop applications: the generated CRUD operations are written in Java language, and visualized with the Eclipse platform, or others. To this end, the transformation rules need to be developed for the target (web) platform. With regard to point 2, given the authors' programming experience, the use of SQL language to query database tables was the logical choice. After reading the documentation, it was found that the use of the Hibernate Query Language (HQL) with the Hibernate ORM is better suited for complex queries and sophisticated data access.

VIII. CONCLUSION AND FUTURE WORK

This paper aimed to introduce a new approach to generate an executable database access code from requirement models and to extend the work presented in [6], and [20]. In these studies, the authors developed transformation algorithms to automatically generate Java code from requirement models, following the Model-View-Presenter architectural pattern. The authors generated code for the Presenter and View layers for both Desktop Application and Web-based application, but they generate only code skeleton for the Model layer. This work focused on the Model layer that is responsible for data persistence in a database system using CRUD operations. To this end, the Hibernate ORM that maps the Data Transfer Objects (DTOs) to Relational database tables was used.

The case study presented in this work for the "Tribunal E-Services" system, was performed using the ReDSeeDS framework to edit the source model which is an RSL model (i.e. using the cases with their stories and the domain model). The transformation rules were written using the MoLA language. The target model composed of model classes in JAVA and Hibernate files was generated by the Enterprise Architect tool. In the near future, the authors intend to present the complete case study associated with the database tables and views. The aim is to use the HQL language to query the tables and perform more complex access operations such as join problem. The authors also intend to merge the source code proposed in this study with the View and Presenter layers code generated in the work available in.[6], [20] and to create an executable web- based application software.

ACKNOWLEDGMENT

The authors would like to thank Dr. Wiktor Nowakowski from Warsaw University of Technology, Poland, for his assistance regarding the ReDSeeDS framework, and MoLA language.

References

[1] B. Cheng, J. Atlee, Research Directions in Requirements Engineering. in Future of Software Engineering, 2007. FOSE '07.
[2] G. Loniewski, A. Armesto, E. Insfran, An architecture-oriented model-driven requirements engineering approach. in Model-Driven Requirements Engineering Workshop (MoDRE), 2011.
[3] I. Jacobson, Use cases - Yesterday, today, and tomorrow, Publisher, City, 2004.
[4] C.D. Manning, H. Schütze, Foundations of Statistical Natural Language Processing1999: MIT Press. 680.
[5] O. Object Management Group®, About the Unified Modeling Language Specification Version 2.5.1, 2017 2020

[cited 2021, [online]. Available: https://www.omg.org/spec/UML/About-UML/.
[6] M. Smiałek, W. Nowakowski, N. Jarzebowski, A. Ambroziewicz, From use cases and their relationships to code. in Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012, Chicago, IL, USA, September 24, 2012.
[7] M. Smiałek, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, Introducing a unified requirements specification language, Publisher, City, 2007.
[8] A.M.R.d. Cruz, A pattern language for use case modeling. in 2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2014.
[9] A. Silva, et al., A Pattern Language for Use Cases Specification, 2015.
[10] A.J. de Souza, A.L.O. Cavalcanti, Visual Language for Use Case Description, Publisher, City, 2016.
[11] B. Berenbach, A 25 year retrospective on model-driven requirements engineering. in Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE.
[12] M. Smialek, W. Nowakowski, From Requirements to Java in a Snap - Model-Driven Requirements Engineering in Practice2015: Springer International Publishing. 352.
[13] H. Kaindl, et al. (2009). *Requirements specification language definition (Deliverable D2.4)*.
[14] ReDSeeDS | Requirements-Driven Software Development System, [cited 2021, [online]. Available: http://smog.iem.pw.edu.pl/redseeds/.
[15] M. Smialek, N. Jarzebowski, W. Nowakowski, Translation of Use Case Scenarios to Java Code, Publisher, City, 2012.
[16] MOLA pages, 2011 2011 [cited 2021, [online]. Available: http://mola.mii.lu.lv/.
[17] C. Bauer, G. King, Hibernate in Action (In Action Series)2004: Manning Publications Co. 408.
[18] S. and Michał, A. Albert, W.N. Tomasz Straszak, Requirements-level language and tools for capturing software system essence, Publisher, City, 2013.
[19] OMG, OMG's MetaObject Facility (MOF) Home Page, [cited 2021, [online]. Available: http://www.omg.org/mof/.
[20] M. Smialek, N. Jarzebowski, W. Nowakowski, Runtime semantics of use case stories. in 2012 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2012, Innsbruck, Austria, September 30 - October 4, 2012.
[21] M. Fowler, GUI Architectures, 2006 [cited 2021, [online]. Available: https://martinfowler.com/eaaDev/uiArchs.html.
[22] A. Kalnins, et al., Building tools by model transformations in Eclipse. in Proceedings of Domain Specific Modeling (DSM) 07. Jyvaskyla University Printing House.
[23] O. Vilitis, A. Kalnins, Technical Solutions for the Transformation-Driven Graphical Tool Building Platform METAclipse, Publisher, City, 2008.
[24] S.S.P. Ltd, ENTERPRISE ARCHITECT, [cited 2021, [online]. Available: https://sparxsystems.com/products/ea/index.html.
[25] Z. Ding, M. Jiang, J. Palsberg, From Textual Use Cases to Service Component Models. in Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems. ACM.
[26] E. Foundation, ATL, [cited 2021, [online]. Available: https://www.eclipse.org/atl/.
[27] S. Sarkar, V.S. Sharma, R. Agarwal, Creating design from requirements and use cases: Bridging the gap between requirement and detailed design. in India Software Engineering Conference, ISEC 2012.
[28] T. Yue, L.C. Briand, Y. Labiche, aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models, Publisher, City, 2015.
[29] M. Mariem, B. Nadia, B.-A. Hanêne, From Language-Independent Requirements to Code Based on a Semantic Analysis. The Twelfth International Conference on Software Engineering Advances, 2017: pp. 145-156.
[30] C. Zolotas, T. Diamantopoulos, K.C. Chatzidimitriou, A.L. Symeonidis, From requirements to source code: a Model-Driven Engineering approach for RESTful web services, Publisher, City, 2017.
[31] O.M. Group, 2021, [online]. Available: https://www.omg.org/mda/.
[32] I. Eclipse Foundation, Acceleo, 2020 [cited 2021, [online]. Available: https://www.eclipse.org/acceleo/.
[33] I. Essebaa, S. Chantit, M. Ramdani, MoDAr-WA: Tool Support to Automate an MDA Approach for MVC Web Application, Publisher, City, 2019.
[34] O.M. Group, About the Semantics Of Business Vocabulary And Business Rules Specification Version 1.5, [cited 2021, [online]. Available: https://www.omg.org/spec/SBVR/About-SBVR/.
[35] OMG, About the MOF Query/View/Transformation Specification Version 1.3, 2021], [online]. Available: https://www.omg.org/spec/QVT/About-QVT/.
[36] I. Eclipse Foundation, Papyrus, [cited 2021, [online]. Available: https://www.eclipse.org/papyrus/.
[37] G. Paolone, M. Marinelli, R. Paesani, P. Di Felice, Automatic Code Generation of MVC Web Applications, Publisher, City, 2020.
[38] L. MKLabs Co., StarUML, [cited 2021, [online]. Available: https://staruml.io/.
[39] C. Hue, D.-H. Dang, USL: A Domain-Specific Language for Precise Specification of Use Cases and Its Transformations, Publisher, City, 2018.
[40] V. Mencl, Procasor : Interactive Environment for Requirement Specification, [cited 2021, [online]. Available: https://d3s.mff.cuni.cz/software/procasor/.
[41] JBehave, [cited 2021, [online]. Available: https://jbehave.org/.
[42] M. Bohlen, AndroMDA Model Driven Architecture Framework - AndroMDA - Homepage, [cited 2021, [online]. Available: http://www.andromda.org/.
[43] e.D. Team, eMoflon, [cited 2021, [online]. Available: https://emoflon.org/.
[44] MODELIO The open source extensible modeling environment supporting: UML, BPMN, ArchiMate, SysML, [online]. Available: https://www.modelio.org/.

Nassima Yamouni-Khelifi is a PhD follower in Computer Science and is a part time Lecturer in the Department of Computer Science at the University of Sciences and Technology -Mohamed Boudiaf-, Oran, Algeria. Contact e-mail address: nassima.yamounikhelifi@univ-usto.dz

Kaddour Sadouni is a Titular Professor and Computer Science Technician. He is the Director of the SIMPA laboratory at the University of Sciences and Technology-Mohamed Boudiaf-, Oran, Algeria. Contact him at kaddour.sadouni@univ-usto.dz

Michał Smiałek is a Titular Professor at Warsaw University of Technology, Warsaw, Poland. He is a researcher in the field of Model-Driven Engineering and Requirements Engineering and has contributed to the ReDSeeDS tool. He is also the leader of the SMoG group. Contact him at smialek@iem.pw.edu.pl or visit https://smialek.iem.pw.edu.pl/

Mahmoud Zennaki is a Senior Lecturer in the Department of Computer Science at the University of Sciences and Technology -Mohamed Boudiaf-, Oran, Algeria. Contact him at mahmoud.zennaki@univ-usto.dz

Contribution of individual authors to the creation of a scientific article

Nassima Yamouni Khelifi has implemented the transformation rules in MoLA, and has written the scenarios of the “Tribunal E-services System” case study, and has realised the comparison between the different approaches.
Kaddour Sadouni has carried out the progress of the Article (as he is the supervisor of the PhD dissertation of Nassima yamouni Khelifi).
Michał Śmiałek has proposed the rules of Section III, and helped in the redaction of the Article, especially in the sections related to the RSL language and ReDSeeDS tool.
Mahmoud Zennaki has carried out the progress of the Article (as he is the co-supervisor of the PhD dissertation of Nassima Yamouni Khelifi)..